Linux/UNIX IPC Programming

Alternative I/O Models: *epoll*

Michael Kerrisk, man7.org © 2025

August 2025

mtk@man7.org

Outline

Rev: # caf166f4161b

Outline

Problems with *poll()* and *select()*

- *poll()* + *select()* are portable, long-standing, and widely used
- But, there are scalability problems when monitoring many FDs, because, on each call:
  1. Program passes a data structure to kernel describing **all** FDs to be monitored
  2. The kernel must recheck **all** specified FDs for readiness
     - This includes hooking (and later unhooking) process to FD wait queues to handle case where it is necessary to block because no FD is ready (can be expensive if many FDs)
  3. The kernel passes a modified data structure describing readiness of **all** FDs back to program in user space
  4. After the call, the program must inspect readiness state of **all** FDs in modified data
- ⇒ Cost of *select()* and *poll()* scales with number of FDs being monitored

man7.org

[TLPI §63.2.5]

Problems with *poll()* and *select()*

- *poll()* and *select()* have a design problem:
  - For many applications, set of monitored FDs is static
    - (Or set changes only slowly)
  - But, kernel doesn't remember monitored FDs between calls
    - ⇒ Info on all FDs must be copied back & forth on each call
- *epoll* improves performance by fixing this design problem
  - Kernel maintains a persistent set of FDs that application is interested in
- *epoll* cost **scales according to number of I/O events**
  - Can give **much better performance when monitoring many FDs!**
    - Especially if #active-FDs << #total-FDs
  - (Signal-driven I/O scales similarly, for same reasons)

[TLPI §63.4.5]

Outline

Overview

- Like *select()* and *poll()*, *epoll* can monitor multiple FDs
- *epoll* returns readiness information in similar manner to *poll()*
- Two main **advantages**:
  - *epoll* can provide **much better performance** when monitoring large numbers of FDs
  - *epoll* provides two **notification modes**: **level-triggered** and **edge-triggered**
    - Default is level-triggered notification
    - *select()* and *poll()* provide only level-triggered notification
    - (Signal-driven I/O provides only edge-triggered notification)
- Linux-specific, since kernel 2.6.0 (2003)

[TLPI §63.4]

epoll instances

Central data structure of *epoll* API is an ***epoll* instance**

- **Persistent** data structure **maintained in kernel space**
  - Referred to in user space via file descriptor
- Can (abstractly) be considered as container for two lists:
  - **Interest list**: list of FDs to be monitored
  - **Ready list**: list of FDs that are ready for I/O
    - Ready list is (dynamic) subset of interest list

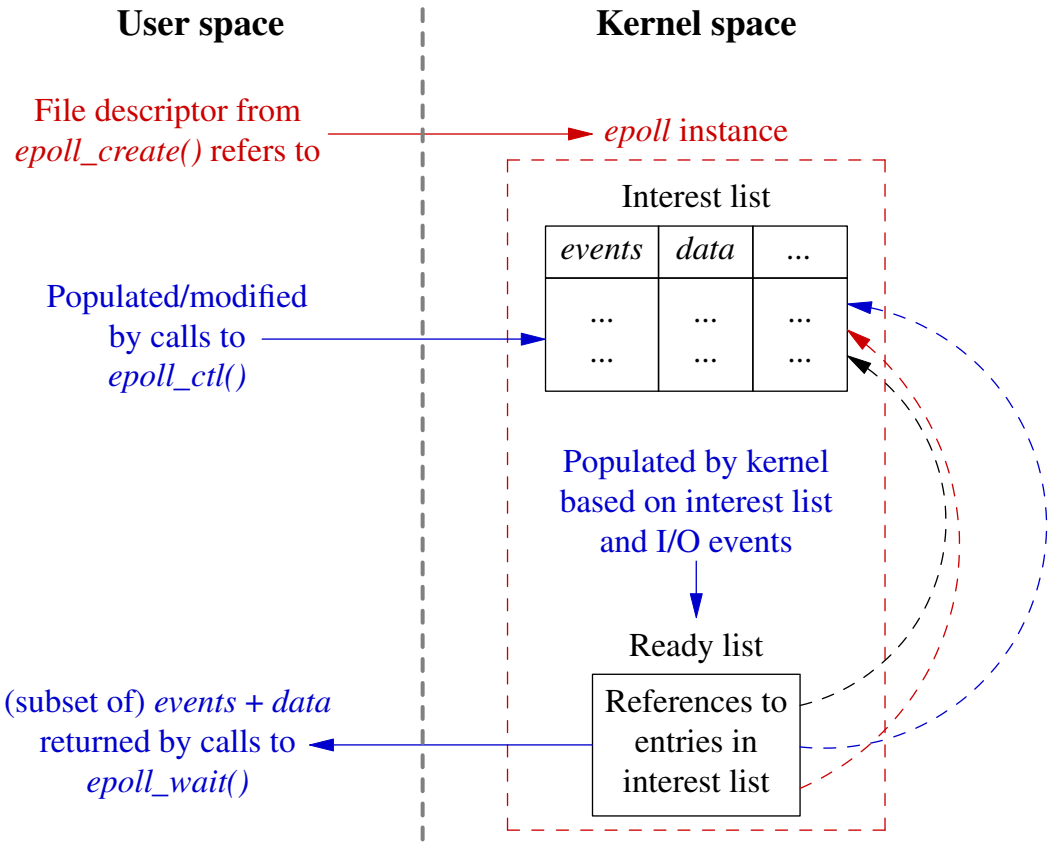
epoll APIs

The key *epoll* APIs are:

- *epoll_create()*: create a new *epoll* instance and return FD referring to instance
  - FD is used in the calls below
- *epoll_ctl()*: modify interest list of *epoll* instance
  - Add FDs to/remove FDs from interest list
  - Modify events mask for FDs currently in interest list
- *epoll_wait()*: return items from ready list of *epoll* instance
- *close()*: close *epoll* FD
  - *epoll* instance torn down if this is last FD referring to instance

man7.org

epoll kernel data structures and APIs



man7.org

Creating an *epoll* instance: *epoll_create()*

```
#include <sys/epoll.h>
int epoll_create(int size);
```

- Creates an *epoll* instance
- *size*:
  - Since Linux 2.6.8: serves no purpose, but must be > 0
    - Backward compatibility: in older kernels, *size==0* resulted in an error, and this behavior has been preserved
  - Before Linux 2.6.8: an *estimate* of number of FDs to be monitored via this *epoll* instance
- Returns file descriptor on success, or –1 on error
  - When FD is no longer required, it should be closed via *close()*
- Since Linux 2.6.27, *epoll_create1()* provides improved API
  - See the manual page

[TLPI §63.4.1]

Modifying the *epoll* interest list: *epoll_ctl()*

```
#include <sys/epoll.h>
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *ev);
```

- Modifies the interest list associated with *epoll* FD, *epfd*
- *fd*: identifies which FD in interest list is to have its settings modified
  - Can be FD for pipe, FIFO, terminal, socket, POSIX MQ
  - Can also be an *epoll* FD
    - An *epoll* FD indicates as readable if ready list is nonempty
  - Can't be FD for a regular file or directory

[TLPI §63.4.2]

epoll_ctl() op argument

The *epoll_ctl() op* argument is one of:

- `EPOLL_CTL_ADD`: add *fd* to interest list
  - *ev* specifies events to be monitored for *fd*
  - If *fd* is already in interest list ⇒ `EEXIST`
- `EPOLL_CTL_MOD`: modify settings of *fd* in interest list
  - *ev* specifies new settings to be associated with *fd*
  - If *fd* is not in interest list ⇒ `ENOENT`
- `EPOLL_CTL_DEL`: remove *fd* from interest list
  - Also removes corresponding entry in ready list, if present
  - *ev* is ignored
  - If *fd* is not in interest list ⇒ `ENOENT`
  - Closing FD automatically removes it from *epoll* interest lists
    - ⚠ But this is not reliable: close does **not** occur in some cases! See later...

The *epoll_event* structure

epoll_ctl() ev argument is pointer to an *epoll_event* structure:

```
struct epoll_event {
    uint32_t     events;  // epoll events (bit mask)
    epoll_data_t data;    // User data
};
typedef union epoll_data {
    void    *ptr;         // Pointer to struct (or function)
    int      fd;          // File descriptor
    uint32_t u32;         // E.g., array index
    uint64_t u64;         // E.g., hash table key
} epoll_data_t;
```

- *ev.events*: bit mask of events to monitor for *fd*
  - (Similar to *events* mask given to *poll()*)
- *data*: info to be passed back to caller of *epoll_wait()* when *fd* later becomes ready
  - **Union field**: value is specified in *one* of the members

Example: using *epoll_create()* and *epoll_ctl()*

```
int epfd = epoll_create(5);

struct epoll_event ev;
ev.data.fd = fd;
ev.events = EPOLLIN;            /* Monitor for readability */

epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &ev);
```

Outline

Waiting for events: *epoll_wait()*

```
#include <sys/epoll.h>
int epoll_wait(int epfd, struct epoll_event *evlist,
               int maxevents, int timeout);
```

- Returns info about ready FDs in interest list of *epoll* instance of *epfd*
- Blocks until at least one FD is ready
- Info about ready FDs is returned in array *evlist*
  - I.e., can get information about multiple ready FDs with one *epoll_wait()* call
  - (Caller allocates the *evlist* array)
- *maxevents*: size of the *evlist* array

[TLPI §63.4.3]

Waiting for events: *epoll_wait()*

```
#include <sys/epoll.h>
int epoll_wait(int epfd, struct epoll_event *evlist,
               int maxevents, int timeout);
```

- *timeout* specifies a timeout for call:
  - –1: block until an FD in interest list becomes ready
  - 0: perform a nonblocking "poll" to see if any FDs in interest list are ready
  - $>$ 0: block for up to *timeout* milliseconds or until an FD in interest list becomes ready
    - *epoll_pwait2()* (Linux 5.11) allows timeout with nanosecond precision
- Return value:
  - $>$ 0: number of items placed in *evlist*
  - 0: no FDs became ready within interval specified by *timeout*
  - –1: an error occurred

Waiting for events: *epoll_wait()*

```
#include <sys/epoll.h>
int epoll_wait(int epfd, struct epoll_event *evlist,
               int maxevents, int timeout);
```

- Info about **multiple** FDs can be returned in the array *evlist*
- Each element of *evlist* returns info about one file descriptor:
  - *events* is a bit mask of events that have occurred for FD
  - *data* is *ev.data* value *currently* associated with FD in the interest list
- **NB**: the FD itself is **not** returned!
  - Instead, we put FD into *ev.data.fd* when calling *epoll_ctl()*, so that it is returned via *epoll_wait()*
    - (Or, put FD into a structure pointed to by *ev.data.ptr*)

Waiting for events: *epoll_wait()*


```
#include <sys/epoll.h>
int epoll_wait(int epfd, struct epoll_event *evlist,
               int maxevents, int timeout);
```


- 👍 If > *maxevents* FDs are ready, successive *epoll_wait()* calls round-robin through FDs
  - Helps prevent file descriptors being starved of attention
- 👍 In multithreaded programs:
  - While one thread is blocked in *epoll_wait()*, another thread can modify interest list (*epoll_ctl()*)
  - *epoll_wait()* call will return if a newly added FD becomes ready

epoll events

Following table shows:

- Bits given in *ev.events* to *epoll_ctl()*
- Bits returned in *evlist[].events* by *epoll_wait()*

Bit	*epoll_ctl()*?	*epoll_wait()*?	Description
EPOLLIN	•	•	Normal-priority data can be read
EPOLLOUT	•	•	Data can be written
EPOLLPRI	•	•	High-priority data can be read
EPOLLRDHUP	•	•	Shutdown on peer socket
EPOLLONESHOT	•		Disable monitoring after event notification
EPOLLET	•		Employ edge-triggered notification
EPOLLHUP		•	A hangup occurred
EPOLLERR		•	An error has occurred

- Other than EPOLLONESHOT and EPOLLET, bits have same meaning as similarly named *poll()* bit flags
- EPOLLIN, EPOLLOUT, EPOLLPRI, and EPOLLRDHUP, are returned by *epoll_wait()* only if specified when adding FD using *epoll_ctl()*

[TLPI §63.4.3]

Example: altio/epoll_read.c

```
./epoll_read file...
```

- Monitors one or more files using *epoll* API to see if input is possible
- Suitable files to give as arguments are:
  - FIFOs
  - Terminal device names
    - (May need to run *sleep* command in foreground on those terminals, to prevent shell stealing input)

Example: altio/epoll_read.c (1)

```
int epfd = epoll_create(argc - 1);

for (j = 1; j < argc; j++) {
    int fd = open(argv[j], O_RDONLY);
    printf("Opened \"%s\" on fd %d\n", argv[j], fd);

    struct epoll_event ev;
    ev.events = EPOLLIN;
    ev.data.fd = fd;
    epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &ev);
}
int numOpenFds = argc - 1;
```

- Create an *epoll* instance, obtaining *epoll* FD
- Open each of the files named on command line
- Monitor each file for input (EPOLLIN)
- Put *fd* into *ev.data*, so it is returned by *epoll_wait()*
- Add the FD to *epoll* interest list (*epoll_ctl()*)
- Track number of open FDs (in *numOpenFds*)

Example: altio/epoll_read.c (2)

```
while (numOpenFds > 0) {
    const int MAX_EVENTS = 5;
    struct epoll_event evlist[MAX_EVENTS];

    printf("About to epoll_wait()\n");
    int ready = epoll_wait(epfd, evlist, MAX_EVENTS, -1);
    if (ready == -1) {
        if (errno == EINTR)
            continue;      /* Restart if interrupted by signal */
        else
            errExit("epoll_wait");
    }

    printf("Ready: %d\n", ready);
```

- Loop, fetching *epoll* events and analyzing results
  - Loop terminates when no more FDs are open
- *epoll_wait()* call places up to MAX_EVENTS events in *evlist*
  - *timeout* $== -1 \Rightarrow$ infinite timeout
- Return value from *epoll_wait()* is number of ready FDs

Example: altio/epoll_read.c (3)

```
    for (int j = 0; j < ready; j++) {
        printf("  fd=%d; events: %s%s\n", evlist[j].data.fd,
                (evlist[j].events & EPOLLIN)  ? "EPOLLIN "  : "",
                (evlist[j].events & EPOLLHUP) ? "EPOLLHUP " : "");

        const int BUF_SIZE = 10;
        char buf[BUF_SIZE];
        ssize_t nr = read(evlist[j].data.fd, buf, BUF_SIZE);
        if (nr == -1)
            errExit("read");
        ...
    }
```

- Iterate through *ready* items in *evlist*
- Display *events* bits for each ready FD
- Read from ready FD
  - Note that we don't even need to check *events*
    - EPOLLIN ⇒ *read()* won't block
    - EPOLLHUP ⇒ *read()* will return 0 (without blocking)

Example: altio/epoll_read.c (4)

```
    for (int j = 0; j < ready; j++) {
        ...
        if (nr == 0) {                /* read() indicated end-of-file */
            printf("    read() returned 0; closing fd %d\n",
                    evlist[j].data.fd);
            epoll_ctl(epfd, EPOLL_CTL_DEL, evlist[j].data.fd, NULL);
            close(evlist[j].data.fd);
            numOpenFds--;
        } else {
            printf("    read %zd bytes: %.*s\n", nr, (int) nr, buf);
        }
    }
}
```

- If *read()* returned 0 (EOF):
  - Remove FD from *epoll* interest list
  - Close FD
- Otherwise, display data that was read
  - %.*s ⇒ field width taken from argument list (*nr*)

Outline

When will *epoll* win?

- Ideal *epoll* use case (vs *select()*/*poll()*):
  - Monitoring "large" number of FDs
  - Only "small" number of FDs are active (ready) at any moment
  - Set of monitored FDs is static (or changes only "slowly")
- Performance of *epoll* vs *poll()*/*select()* is similar if:
  - Set of FDs being monitored is "small"
- Performance of *poll()*/*select()* may even surpass *epoll* if:
  - At each monitoring step, "many" FDs are ready, or
  - Set of FDs being monitored changes "frequently"
    - Because of cost of *epoll_ctl()* syscalls to add/remove FDs from interest list

Timings: notes and caveats

- Measurements using `altio/altio_speed.c`
- Simple model; single process, performing steps as follows:
  - Create multiple pipes (or sockets) †
  - Write data to a certain number of pipes
  - Measure time required for loop that repeatedly calls *select()*, *poll()*, or *epoll_wait()*
- ⇒ Numbers on next slides are **very simplistic** benchmarks
  - Real world servers might, e.g., split load across thread pool
  - No measurement of network latency effects or cost of I/O system calls or data processing work

† The program uses *dup2()* to arrange the read ends of the pipes to be contiguous FDs at the low end of the number range; this maximizes the number of FDs that can be monitored with *select()* and minimizes the size of the *select()* *readfds* argument

man7.org

Timings: sparsely active file descriptors

# monitored FDs (*N*)	Elapsed time (seconds)		
	select()	*poll()*	*epoll*
1	0.23	0.19	0.16
10	0.48	0.46	0.16
100	2.78	3.18	0.16
1000	30.0	36.4	0.16

- 1 active FD, at mid-point in list of FDs; 1'000'000 monitoring operations
  - `./altio_speed -f N -l 1000000 -M mode`
- When the ratio of active vs total FDs is low, *epoll* clearly wins
- But this reverses when the ratio is sufficiently large...

man7.org

Timings: varying density of ready file descriptors

# monitored FDs (*N*)	# ready FDs (*R*)	*R*/*N*	Elapsed time (seconds)		
			select()	*poll()*	*epoll*
100	10	0.1	1.09	1.20	0.55
100	20	0.2	0.99	1.04	0.99
100	50	0.5	0.88	0.96	2.3
100	100	1.0	0.86	0.94	4.4
1000	100	0.1	13.3	14.0	4.5
1000	200	0.2	13.2	14.0	9.4
1000	500	0.5	12.9	13.5	23.6
1000	1000	1.0	11.8	12.7	47.3

- Ready FDs randomly distributed; 1'000'000 monitoring operations
  - `./altio_speed -f N -l 1000000 -w R mode`
- Times *decrease* for *select()* + *poll()* with larger numbers of ready FDs
  - Hypothesis: more ready FDs == shorter search time until first ready FD is found ⇒ fewer FDs need to be hooked + unhooked
    - (See exercise 2 in following slides)

Timings: effect of interest list changes on *epoll*

# FDs	# interest list changes (*K*)	Elapsed time (seconds)
100	0	0.16
100	1	0.69
100	5	2.8
100	10	5.4

- 100 FDs, 1 active FD, at mid-point in list of FDs; 1'000'000 monitoring operations
  - `./altio_speed -f 100 -l 1000000 -M -e K e`
- In each loop, *K* `EPOLL_CTL_DEL` and *K* `EPOLL_CTL_ADD` operations are performed
  - I.e., *2 * K epoll_ctl()* syscalls / loop
    - And syscalls are expensive...

Outline

API summary

```
// Create an epoll instance, returning file descriptor:
int epoll_create(int size);

// Modify epoll interest list:
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *ev);
    // 'op' is EPOLL_CTL_ADD / EPOLL_CTL_MOD / EPOLL_CTL_DEL

// Structure passed to epoll_ctl() / returned by epoll_wait():
struct epoll_event {
    unint32_t    events;
    epoll_data_t data;
};
typedef union epoll_data {
    void *ptr;
    int fd;
    uint32_t u32;
    uint64_t u64;
};

// Fetch events from epoll ready list, returning then in 'evlist':
int epoll_wait(int epfd, struct epoll_event *evlist, int maxevents,
               int timeout);
    // 'timeout' can be: >0: milliseconds; 0: don't block; -1: block indefinitely
```

man7.org

Outline

Exercises

1. Write a client ([template: `altio/ex.is_chat_cl.c`]) that communicates with the TCP chat server program, `is_chat_sv.c`. The program should be run with the following command line:

```
./is_chat_cl <host> <port> [<nickname>]
```

The program should create a connection to the server, and then use the *epoll* API to monitor both the terminal and the TCP socket for input. All input that becomes available on the socket should be written to the terminal and vice versa.

- Each time the program sends input from the terminal to the socket, that input should be prepended by the nickname supplied on the command line. If no nickname is supplied, then use the string returned by *getlogin(3)*. (*snprintf(3)* provides an easy way to concatenate the strings.)

[Exercise continues on next slide]

Exercises

- Both the terminal and the socket will indicate as readable (EPOLLIN) when input becomes available or when an end-of-file condition occurs.
- The program should terminate if it detects end-of-file on either file descriptor.
- Calling *epoll_wait()* with *maxevents==1* will simplify the code!

```
struct epoll_event rev;
epoll_wait(epfd, &rev, 1, -1);
```

(This is simpler, because then you don't have to iterate through an array that would in any case contain at most two entries.)

- As a simplification, you can assume that the socket is always writable (i.e., you don't need to monitor for the socket for EPOLLOUT).
- Bonus points if you find a way to crash the server (reproducibly)!

man7.org

Exercises

2. Consider the observation on slide 12-31, that when large numbers of file descriptors are ready, then *poll()* and *select()* are faster, perhaps because fewer FD wait queues need to be hooked and unhooked. We can explore this hypothesis by comparing the timing measurements from two different executions of the altio_speed.c program:

```
$ cd lsp/altio
$ sudo prlimit --nofile=10000:10000 --pid=$$      # Raise shell's FD limit
$ time ./altio_speed -f 1000 -l 100000 -x 999 p
$ time ./altio_speed -f 1000 -l 100000 -x 0 p
```

In the above, 1000 FDs are monitored 100'000 times using *poll()*, and just one of the FDs is ready. In the first case, the ready FD is at the end of the *pollfd* array; consequently, the kernel needs to hook (and later unhook) the process on wait queues for the 999 preceding FDs. In the second case, the ready FD is at the start of the array; thus, the kernel knows that it will not be necessary for the *poll()* call to block, so that no hooking will need to be done for any of the remaining FDs.

Run the two commands. Is there a notable difference in the displayed timings? Repeat the experiment for *select()* (change *p* to *s*).

man7.org